Rust Book - 9.0: Error Handling

#rust Error Handling

Rust's commitment to reliability extends to error handling. Errors are a fact of life in software, so Rust has a number of features for handling situations in which something goes wrong.

In many cases, Rust requires you to acknowledge the possibility of an error and take some action before your code will compile. This requirement makes your program more robust by ensuring that you'll discover errors and handle them appropriately before you've deployed your code to production!

Rust groups errors into two major categories: recoverable and unrecoverable errors.

For a recoverable error, such as a file not found error, it's reasonable to report the problem to the user and retry the operation.

Unrecoverable errors are always symptoms of bugs, like trying to access a location beyond the end of an array.

Most languages don't distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. Rust doesn't have exceptions.

Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error.

Unwinding the Stack or Aborting in Response to a Panic

Unwinding the Stack or Aborting in Response to a Panic

By default, when a panic occurs, the program starts **unwinding**, which means Rust walks back up the stack and cleans up the data from each function it encounters. But this walking back and cleanup is a lot of work. The alternative is to immediately **abort** , which ends the program without cleaning up.

Memory that the program was using will then need to be cleaned up by the operating system. For smaller binary: if you want to abort on panic in release mode, add this:

```
[profile.release]
panic = ‘abort’
```

Using a `panic!` Backtrace

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}



$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
     Running `target/debug/panic`
thread ‘main’ panicked at ‘index out of bounds: the len is 3 but the index is
99’, /checkout/src/liballoc/vec.rs:1555:10
```

Other languages, like C, will attempt to give you exactly what you asked for in this situation, even though it isn’t what you want: you’ll get whatever is at the location in memory that would correspond to that element in the vector, even though the memory doesn’t belong to the vector. This is called a buffer overread and can lead to security vulnerabilities if an attacker is able to manipulate the index in such a way as to read data they shouldn’t be allowed to that is stored after the array.

To protect your program from this sort of vulnerability, if you try to read an element at an index that doesn’t exist, Rust will stop execution and refuse to continue.

Recoverable Errors withResult

Most errors aren’t serious enough to require the program to stop entirely. Sometimes, when a function fails, it’s for a reason that you can easily interpret and respond to. e.g. File read operations

```
let f = File::open("hello.txt");
let f = match f {
    Ok(file) => file,
    Err(error) => {
        panic!("Problem opening the file: {:?}!", error)
    },
};
```

Matching on Different Errors

```
let f = File::open("hello.txt");

let f = match f {
    Ok(file) => file,
    Err(error) => match error.kind() {
        ErrorKind::NotFound => match File::create("hello.txt") {
            Ok(fc) => fc,
            Err(e) => panic!("Tried to create file but there was a problem:
{:?}", e),
        },
        ErrorKind::PermissionDenied => panic!("Dont got permissions: {:?}",
error),
        other_error => panic!("There was a problem opening the file: {:?}",
other_error),
    },
};
```

Shortcuts for Panic on Error:unwrapandexpect

`unwrap` , is a shortcut method that is implemented just like the `match` . If the `Result` value is the `Ok` variant, `unwrap` will return the value inside the `Ok` . If the `Result` is the `Err` variant, `unwrap` will call the `panic!` macro for us.

`expect` lets us also choose the error message.

```
let f = File::open("hello.txt").unwrap();
// OR
let f = File::open("hello.txt").expect("Failed to open hello.txt");
```

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code:
2, message: "No such file or directory" } }', src/libcore/result.rs:906:4
```

Because this error message starts with the text we specified, `Failed to open hello.txt`, it will be easier to find where in the code this error message is coming from. If we use `unwrap` in multiple places, it can take more time to figure out exactly which `unwrap` is causing the panic because all `unwrap` calls that panic print the same message.

Propagating Errors

Propagating Errors

When you're writing a function whose implementation calls something that might fail, instead of handling the error within this function, you can return the error to the calling code so that it can decide what to do. This is known as propagating the error and gives more control to the calling code, where there might be more information or logic that dictates how the error should be handled than what you have available in the context of your code.

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");
    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

return type of the function first: `Result<String, io::Error>.` This means the function is returning a value of the type `Result<T, E>` where the generic parameter `T` has been filled in with the concrete type `String`, and the generic type `E` has been filled in with the concrete type `io::Error`.

A Shortcut for Propagating Errors: the `?` Operator

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
match read_username_from_file() {
    Ok(s) => println!("Hello {}!", s),
    Err(e) => println!("Error!!!! {:?}", e),
}
```

There is a difference between what the `match` expression and `?` do: error values taken by `?` go through the `from` function, defined in the `From` trait in the standard library, which is used to convert errors from one type into another.

When `?` calls the `from` function, the error type received is converted into the error type defined in the return type of the current function.

```
let mut s = String::new();
File::open("hello.txt")?.read_to_string(&mut s)?;
Ok(s)
```

```rust
use std::io;
use std::fs;
fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

The?Operator Can Only Be Used in Functions That Return `Result`

The?operator can only be used in functions that have a return type of `Result` , because it is defined to work in the same way as thematchexpression

The part of thematchthat requires a return type of `Result` isreturn Err(e), so the return type of the function must be a `Result` to be compatible with thisreturn.

In functions that don't return `Result` <T, E>, when you call other functions that return `Result` <T, E>, you'll need to use amatchor one of the `Result` <T, E>methods to handle the `Result` <T, E>instead of using?to potentially propagate the error to the calling code.

```rust
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;
    Ok(())
}
```

To `panic!` or Not to `panic!`

To `panic!` or Not to `panic!`

When code panics, there's no way to recover. You could call `panic!` for any error situation, whether there's a possible way to recover or not, but then you're making the decision on behalf of the code calling your code that a situation is unrecoverable.

When you choose to return a `Result` value, you give the calling code options rather than making the decision for it. The calling code could choose to attempt to recover in a way that's appropriate for its situation, or it could decide that an `Err` value in this case is unrecoverable, so it can call `panic!` and turn your recoverable error into an unrecoverable one.

Returning `Result` is a good default choice when you're defining a function that might fail.

Cases in Which You Have More Information Than the Compiler

Cases in Which You Have More Information Than the Compiler

It would also be appropriate to callunwrapwhen you have some other logic that ensures the `Result` will have anOkvalue, but the logic isn't something the compiler understands. You'll still have a `Result` value that you need to handle: whatever operation you're calling still has the possibility of failing in general, even though it's logically impossible in your particular situation. If you can ensure by manually inspecting the code that you'll never have anErrvariant, it's perfectly acceptable to callunwrap.

```
use std::net::IpAddr;
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

We're creating an `IpAddr` instance by parsing a hardcoded string. We can see that `127.0.0.1` is a valid IP address, so it's acceptable to useunwraphere. However, having a hardcoded, valid string doesn't change the return type of theparsemethod: we still get a `Result` value, and the compiler will still make us handle the `Result` as if theErrvariant is a possibility because the compiler isn't smart enough to see that this string is always a valid IP address. If the IP address string came from a user rather than being hardcoded into the program and therefore **did** have a possibility of failure, we'd definitely want to handle the `Result` in a more robust way instead.

Guidelines for Error Handling

Guidelines for Error Handling

It's advisable to have your code panic when it's possible that your code could end up in a bad state. In this context, a **bad state** is when some assumption, guarantee, contract, or invariant has been broken, such as when invalid values, contradictory values, or missing values are passed to your code—plus one or more of the following:

- The bad state is not something that's **expected** to happen occasionally.
- Your code after this point needs to rely on not being in this bad state.

- There’s not a good way to encode this information in the types you use.

If someone calls your code and passes in values that don’t make sense, the best choice might be to call `panic!` and alert the person using your library to the bug in their code so they can fix it during development. Similarly, `panic!` is often appropriate if you’re calling external code that is out of your control and it returns an invalid state that you have no way of fixing.

However, when failure is expected, it is more appropriate to return a `Result` than to make a `panic!` call. Examples include a parser being given malformed data or an HTTP request returning a status that indicates you have hit a rate limit. In these cases, returning a `Result` indicates that failure is an expected possibility that the calling code must decide how to handle.

When your code performs operations on values, your code should verify the values are valid first and panic if the values aren’t valid. This is mostly for safety reasons: attempting to operate on invalid data can expose your code to vulnerabilities. This is the main reason the standard library will call `panic!` if you attempt an out-of-bounds memory access: trying to access memory that doesn’t belong to the current data structure is a common security problem.

Functions often have **contracts**: their behavior is only guaranteed if the inputs meet particular requirements. Panicking when the contract is violated makes sense because a contract violation always indicates a caller-side bug and it’s not a kind of error you want the calling code to have to explicitly handle. In fact, there’s no reasonable way for calling code to recover; the calling **programmers** need to fix the code. Contracts for a function, especially when a violation will cause a panic, should be explained in its API documentation.

However, having lots of error checks in all of your functions would be verbose and annoying. You can use Rust’s type system (and thus the type checking the compiler does) to do many of the checks for you.

If your function has a particular type as a parameter, you can proceed with your code’s logic knowing that the compiler has already ensured you have a valid value. e.g:

- if you have a type rather than anOption, your program expects to have*something*rather than*nothing*. Your code then doesn’t have to handle two cases for the `Some` and `None` variants: it will only have one case for definitely having a value. Code trying to pass nothing to your function won’t even compile, so your function doesn’t have to check for that case at runtime.

- using an unsigned integer type such as `u32`, which ensures the parameter is never negative.

```
pub struct Guess {
    value: i32,
}
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("must be >1 and <100, got {}.", value);
        }
        Guess { value }
    }
    pub fn value(&self) -> i32 {
        self.value
    }
}
```